FFT についてのメモ

— FFTPACK の利用に向けて —

桂田 祐史

1996 年 10 月 15 日, 2004 年 1 月, 2016 年 12 月 (久しぶり訂正)

この文書に何か誤りを見つけた場合は、筆者まで連絡して頂けると幸いです。連絡先: mk@math.meiji.ac.jp

目 次

1 はじめに

以下この文書では**高速 Fourier 変換** (fast Fourier transform, 一般に **FFT** と略称される) について、利用者の立場からの解説を行う。

FFT とは、**離散 Fourier 変換** (discrete Fourier transform, 以下では **DFT** と略記する) を非常に効率的に計算するためのアルゴリズム[1]である。

DFT が何であるかについては、2 節で詳しく解説するが、手短にいうと、次の (i), (ii) の総称である (ゆえに DFT は有限次元数ベクトル空間上の一次変換の一種である)。

(i) 周期関数の、一周期区間の等分割点上での関数値からなる数列に、その関数の Fourier 係数の近似値[2]を対応させる写像

(ii) 周期関数の Fourier 係数に、その関数の一周期区間の等分割点上での関数値[3]からなる数列を対応させる写像

Fourier 級数、Fourier 変換が重要であることから、その離散版である DFT も重要なのはもちろんであるが、FFT が驚異的に効率的なアルゴリズムであるために、普通ならば、わざわざ DFT に帰着させて計算しよう、とまでは考えないような問題にも応用されるようになった。FFT の登場は、まさに「(大きな) 量 (の変化) は質を変える」を地で行くものと言えよう。

さまざまな意味で FFT は学ぶに価するものであるが、この稿では、

- DFT の意味 (定義と基本的な性質)
- FFTPACK という FFT ライブラリィの使い方

のみを解説し、

- FFT では実際にどうやって計算するのか
- FFT はなぜ効率的なのか

などの説明、すなわちアルゴリズムとその効率の解析についての詳しい説明は省略する。つまり、FFT は何に役立つのか、どうやって使えばいいのか、ということに焦点を絞る。

参考書としては、例えば

1. 森正武・名取亮・鳥居達生、岩波講座情報科学 18 数値計算、第 5 章、岩波書店
2. 現代応用数学の基礎 1 、第 1.6 節「高速フーリエ変換」、日本評論社
3. William H. Press 他、ニューメリカル・レシピ・イン・シー、第 12 章、技術評論社

等をあげておく。FFT のアルゴリズムの解説については、これらの文献を参照してもらうことを期待している。

一数学者としての所感: FFT は、他の数値解析の手法に比べると、登場してから比較的日が浅いためか[4]、特に数学者村において、知名度は今ひとつ、というところがある[5]。これは離散 Fourier 変換そのものは、その定義から計算法は自明であるし、効率的なアルゴリズムの追求などは本質的な問題ではない、と考えるためであろうか[6]。

[1]数学者の感覚からすると DFT は写像であるのに、FFT はアルゴリズム、何かつりあいが取れないような感じもするが、工学の世界では、「変換」は「○○を計算すること or そのための方法」という程度のとらえ方で、あまり堅苦しく考えないものなのだろう (?)。

[2]Fourier 係数の定義式の定積分を台形則で近似したもの。

[3]一般には無限級数になる Fourier 級数を、有限項までしか計算していない、という意味では近似計算であるが、有限 Fourier 級数の計算としては、丸め誤差がない限り正確な値が得られる方法である。

[4]と言っても、一般に認知されるきっかけとなった Cooley & Tukey の発表が 1965 年だから、もう三十年くらい経っていることになる。

[5]今、手元にある数値解析の本を何冊かめくって見ても、取り上げているものはあまり多くない。

[6]同じことが連立一次方程式の解法にも言える。これは数値解析の重要な話題であるにも関らず、興味を持つ数学者は少ない。

ひさしぶりに (2004 年 1 月)　ふと思い出して、約 8 年ぶりにこの文書を引っ張り出した。必要があって FFTPACK を使ったのが約 10 年前。浦島太郎になったような感じがする。今だったら、大浦 [1] などから必要なものを取得するのかも知れない。

2　DFT に対する数学的説明

ここでは写像としての DFT の定義と簡単な性質を述べる。この節を通じて、文字 i で虚数単位 $\sqrt{-1}$ を表す。

2.1　複素数値関数の DFT

以下 S^1 で $\mathbf{R}/2\pi\mathbf{Z}$ を表す。つまり、実軸 $\mathbf{R}$ 上で定義された、周期 2π の関数と言うかわりに、S^1 上の関数というわけである。

$f\colon S^1 \to \mathbf{C}$ が適当な滑らかさをもつ時、次のように Fourier 級数展開できる:

$$f(x) = \sum_{k\in\mathbf{Z}} \widehat{f}(k) e^{ikx} \quad (x \in S^1). \tag{1}$$

ここで、$\widehat{f}(k)$ は f の Fourier 係数と呼ばれるもので、次式で定義される:

$$\widehat{f}(k) = \frac{1}{2\pi}\int_0^{2\pi} f(x)e^{-ikx}\,dx \quad (k \in \mathbf{Z}). \tag{2}$$

さて、自然数 N を一つ固定して、S^1 上の点 x_j を

$$x_j = \frac{2\pi j}{N} \quad (j \in \mathbf{Z}) \tag{3}$$

で定め、関数 f の x_j での値を f_j とおく:

$$f_j = f(x_j) \quad (j \in \mathbf{Z}). \tag{4}$$

こうして定めた $\{f_j\}$ は、周期 N の周期数列であるので、$j = 0, 1, \cdots, N-1$ について考えれば十分である (つまり S^1 の N 等分点上の関数値を考えることになる)。

今 $\{f_j; j = 0, 1, \cdots, N-1\}$ が与えられたとして、f の Fourier 係数 $\widehat{f}(k)$ を計算することを考えよう。それには、(2) の積分を数値積分で求めるのが自然であるが、実は

周期関数の一周期区間における定積分を数値積分するには台形則が最適である[7]

という事実があるので、次式で定義される $\widehat{f^a}(k)$ を $\widehat{f}(k)$ の近似値として採用するのがよいと考えられる:

$$\widehat{f^a}(k) \stackrel{\text{def.}}{\equiv} \frac{1}{2\pi}\sum_{j=0}^{N-1}\frac{2\pi}{N} f(x_j)\exp\left(-ik\frac{2\pi j}{N}\right) = \frac{1}{N}\sum_{j=0}^{N-1} f_j \exp\left(-\frac{2\pi i k j}{N}\right) = \frac{1}{N}\sum_{j=0}^{N-1} f_j \omega^{-kj}, \tag{5}$$

ただし、

$$\omega = \exp\frac{2\pi i}{N}. \tag{6}$$

この近似 Fourier 係数 $\widehat{f^a}(k)$ については、次が成り立つ。

[7] これがどういう意味であるかの説明は省略する。数値解析の教科書を参照されたい。例えば、森正武「数値解析」共立出版。

定理 2.1　1. 近似 Fourier 係数は、本来の Fourier 係数によって次のように書ける。

$$\widehat{f}^a(k) = \sum_{p\in\mathbf{Z}} \widehat{f}(k+pN) = \sum_{\ell\equiv k \pmod N} \widehat{f}(\ell) \quad (k\in\mathbf{Z}). \tag{7}$$

2. (近似 Fourier 係数の周期性)

$$k \equiv \ell \pmod N \Longrightarrow \widehat{f}^a(k) = \widehat{f}^a(\ell). \tag{8}$$

補題 2.2 $p\in\mathbf{Z}$ に対して

$$\sum_{j=0}^{N-1} \omega^{pj} = \begin{cases} N & (p\equiv 0 \pmod N) \\ 0 & (\text{それ以外}). \end{cases}$$

定理の証明　（略）■

(8) から、近似 Fourier 係数は、連続する N 項 $\{f_j; j=0,1,\cdots,N-1\}$ について考えれば十分であることが分かる。(7) も含めて考えると、次のようなことが言えるだろう。

- j が 0 に近いところでは、$\widehat{f}^a(j)$ は $\widehat{f}(j)$ の良い近似である。
- j が $N-1$ に近いところでは、$\widehat{f}^a(j)$ は $\widehat{f}(j)$ よりはむしろ $\widehat{f}(j-N)$ の近似であると考えるべきで、近似の程度も良い。
- j が $N/2$ に近いところでは、$\widehat{f}^a(j) \approx \widehat{f}(j)+\widehat{f}(j-N)$ で、右辺の二つの項はほとんど同じ程度の量であると期待できるから (大体どちらを近似しているとも言えないほど)、近似の程度は悪い。

注意　次のような事実があるので、絶対値の大きな k に対する $|\widehat{f}(k)|$ は小さい、と考えよう。

1. Riemann-Lebesgue の定理: $f\in L^1$ ならば $\widehat{f}(k) = o(1)$ (as $k\to\pm\infty$).
2. f が C^m 級ならば $\widehat{f}(k) = o(|k|^{-m})$ (as $k\to\pm\infty$).
3. f が解析的ならば $k\to\pm\infty$ のとき $\widehat{f}(k)$ は指数的に減少する。

定理 2.3 (複素 DFT) (5) により $\widehat{f}^a(k)$ を定めるとき、写像

$$\mathcal{F}\colon \mathbf{C}^N \ni \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{N-1} \end{pmatrix} \longmapsto \begin{pmatrix} \widehat{f}^a(0) \\ \widehat{f}^a(1) \\ \vdots \\ \widehat{f}^a(N-1) \end{pmatrix} \in \mathbf{C}^N \tag{9}$$

は全単射であり、その逆写像は

$$f_j = \sum_{k=0}^{N-1} \widehat{f}^a(k)\omega^{kj} \quad (j=0,1,\cdots,N-1) \tag{10}$$

で与えられる。

証明　$\mathcal{F}$ の表現行列は

$$\frac{1}{N}\begin{pmatrix} \omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 \\ \omega^0 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(N-1)} \\ \omega^0 & \omega^{-2} & \omega^{-4} & \cdots & \omega^{-2(N-1)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ \omega^0 & \omega^{-(N-1)} & \omega^{-(N-1)2} & \cdots & \omega^{-(N-1)(N-1)} \end{pmatrix} \tag{11}$$

である。これが正則行列であり、その逆行列が

$$\begin{pmatrix} \omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \cdots & \omega^{(N-1)} \\ \omega^0 & \omega^2 & \omega^4 & \cdots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ \omega^0 & \omega^{(N-1)} & \omega^{(N-1)2} & \cdots & \omega^{(N-1)(N-1)} \end{pmatrix} \tag{12}$$

であることを示せばよいが、これは

$$\frac{1}{N}\sum_{j=0}^{N-1}\omega^{(m-n)j} = \delta_{0,m-n} \quad (0 \le m, n \le N-1) \tag{13}$$

より分かる[8]。 ■

上の (9) で定義される写像を (N 項) 離散 (型) Fourier 変換 (略して DFT)、その逆写像を離散逆 Fourier 変換 (略して逆 DFT) と呼ぶ。前者を順方向の DFT、後者を逆方向の DFT と呼んだり、両方まとめて DFT と総称することもある。

後で紹介する FFTPACK では、これを実行するために **cffti**, **cfftf**, **cfftb** という手続きが用意されている。

2.2　実数値関数の DFT

ここでは f は実数値関数、すなわち $f\colon S^1 \to \mathbf{R}$ とする。もちろん実数値関数は複素数値関数でもあるから、2.1 のやり方でも計算できるが、次のように係数に冗長性が現れ、実際の計算の際には効率上かなりの損をすることになる:

補題 2.4 (実数値関数の Fourier 係数の Hermite 対称性)

$$f \text{ が実数値関数} \Longrightarrow \widehat{f}(-k) = \overline{\widehat{f}(k)}, \quad \widehat{f^a}(-k) = \overline{\widehat{f^a}(k)} \quad (k \in \mathbf{Z}). \tag{14}$$

さらに、C 言語など、複素数を基本データ型としてサポートしていない言語でプログラミングすることも考えると、実数のまま処理する手続きを用意することが望ましいと分かる。

実数値周期関数は三角関数による Fourier 級数展開で扱うのがよい。まず、その復習から始めよう。$f\colon S^1 \longrightarrow \mathbf{R}$ が適当な滑らかさを持つならば、

$$f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty}(a_k \cos kx + b_k \sin kx) \tag{15}$$

と Fourier 級数展開される。ここで a_k, b_k は Fourier 係数と呼ばれ、次式で定義される:

$$a_k = \frac{1}{\pi}\int_0^{2\pi} f(x)\cos kx\, dx \quad (k = 0, 1, \cdots) \tag{16}$$

$$b_k = \frac{1}{\pi}\int_0^{2\pi} f(x)\sin kx\, dx \quad (k = 1, 2, \cdots). \tag{17}$$

[8] ちょっと気がつきにくいが、(13) は公比 ω^{m-n} の等比級数である。

補題 2.5 (複素 Fourier 係数との関係)

$$\widehat{f}(k) = \frac{1}{2}(a_k - ib_k) \quad (k \in \mathbf{N}), \quad \widehat{f}(0) = \frac{1}{2}a_0. \tag{18}$$

複素 DFT と同様に、積分を台形則で近似すると、a_k, b_k の近似値として、それぞれ次式で定義される A_k, B_k を採用することになる:

$$A_k \equiv \frac{1}{\pi}\sum_{j=0}^{N-1} f_j \cos\left(k \cdot \frac{2\pi j}{N}\right) \cdot \frac{2\pi}{N} = \frac{2}{N}\sum_{j=0}^{N-1} f_j \cos\frac{2\pi kj}{N}, \tag{19}$$

$$B_k \equiv \frac{1}{\pi}\sum_{j=0}^{N-1} f_j \sin\left(k \cdot \frac{2\pi j}{N}\right) \cdot \frac{2\pi}{N} = \frac{2}{N}\sum_{j=0}^{N-1} f_j \sin\frac{2\pi kj}{N}. \tag{20}$$

ここで次の事実を注意しておく:

補題 2.6 (近似 Fourier 係数の性質) 1. (複素近似 Fourier 係数との関係)

$$\widehat{f^a}(k) = \frac{1}{2}(A_k - iB_k) \quad (k \in \mathbf{N}), \quad \widehat{f^a}(0) = \frac{1}{2}A_0. \tag{21}$$

2. (近似 Fourier 係数の周期性)

$$k \equiv \ell \pmod N \Longrightarrow A_k = A_\ell, B_k = B_\ell. \tag{22}$$

3. (近似 Fourier 係数の対称性)

$$A_{N-k} = A_k, B_{N-k} = -B_k. \quad \text{特に } N \text{ が偶数} \Longrightarrow B_{N/2} = 0. \tag{23}$$

定理 2.7 (実 DFT) N の偶奇で場合わけする。

N **が偶数の場合** 写像

$$(f_0, f_1, \cdots, f_{N-1}) \longmapsto (A_0, A_1, B_1, A_2, B_2, \cdots, A_{N/2-1}, B_{N/2-1}, A_{N/2}) \tag{24}$$

は $\mathbf{R}^N$ から $\mathbf{R}^N$ への全単射で、逆写像は

$$f_j = \frac{A_0}{2} + \sum_{k=1}^{N/2-1}\left(A_k \cos\frac{2\pi kj}{N} + B_k \sin\frac{2\pi kj}{N}\right) + A_{N/2}(-1)^j \quad (j = 0, 1, \cdots, N-1) \tag{25}$$

で与えられる。

N **が奇数の場合** 写像

$$(f_0, f_1, \cdots, f_{N-1}) \longmapsto (A_0, A_1, B_1, A_2, B_2, \cdots, A_{(N-1)/2}, B_{(N-1)/2}) \tag{26}$$

は $\mathbf{R}^N$ から $\mathbf{R}^N$ への全単射で、逆写像は

$$f_j = \frac{A_0}{2} + \sum_{k=1}^{(N-1)/2}\left(A_k \cos\frac{2\pi kj}{N} + B_k \sin\frac{2\pi kj}{N}\right) \quad (j = 0, 1, \cdots, N-1) \tag{27}$$

で与えられる。

上の (24), (25), (26),(27) で定義される写像 $\mathbf{R}^N \ni (f_j) \mapsto (A_k, B_k) \in \mathbf{R}^N$ は (N 項) 実 DFT と呼ばれる。

FFTPACK では、この問題を扱うために、二系統の手続きを用意している。

(i) 手続き名の先頭が "ezfft" である[9]三つの手続き **ezffti**, **ezfftf**, **ezfftb** の組

(ii) 手続き名の先頭が "rfft" である[10]三つの手続き **rffti**, **rfftf**, **rfftb** の組

2.3 実数値偶関数に対する DFT — 離散余弦変換

ここでは 2.2 の記号を引き続き用いる。

関数 $f\colon S^1 \to \mathbf{R}$ が偶関数である、すなわち

$$f(-x) = f(x) \quad (x \in S^1) \tag{28}$$

を満足すると仮定すると、Fourier 係数のうち sin の項に対応するものは 0 になる:

$$b_k = 0 \quad (k \in \mathbf{Z}). \tag{29}$$

N 項実 DFT を考えよう。f が偶関数であることから、関数値の作る列 $\{f_j\}$ について、

$$f_j = f_{N-j} \quad (j \in \mathbf{Z}) \tag{30}$$

が成り立つ[11]ことに注意すると、K を

$$K = \begin{cases} N/2 - 1 & (N \text{ が偶数の場合}) \\ (N-1)/2 & (N \text{ が奇数の場合}) \end{cases} \tag{31}$$

で定めて

$$(f_0, f_1, \cdots, f_K) \longmapsto (A_0, A_1, \cdots, A_K) \tag{32}$$

という写像を扱うと考えるのがよい、ということが分かる。(24), (25), (26), (27) から

$$\begin{aligned} A_k &= \frac{2}{N} \sum_{j=0}^{N-1} f_j \cos \frac{2\pi kj}{N} \\ &= \frac{2}{N} \left(f_0 + 2 \sum_{j=1}^{K} f_j \cos \frac{2\pi kj}{N} + \begin{cases} (-1)^k f_{N/2} & (N\colon \text{even}) \\ 0 & (N\colon \text{odd}) \end{cases} \right). \end{aligned} \tag{33}$$

半分の添字 $j = 0, 1, \cdots, K$ しか使わないですむ理由は、「偶関数では、半周期分の関数値のデータを得れば、全体が再生出来る」ということで、理解しやすい (当たり前のこと)。実際の応用では、半周期区間を等分するということで、ほとんどすべての場合に N は偶数である[12]。このとき、

$$N_h \overset{\text{def.}}{\equiv} \frac{N}{2} \tag{34}$$

---

[9] "ez" は "easy" の略であろう。

[10] "r" は "real" の略であろう。

[11] $f_{-j} = f_j$ と書いた方が偶関数の定義 (28) に即しているが、$\{0, 1, \cdots, N-1\}$ の範囲で考える場合には、(30) の方が分かりやすいだろう。

[12] 例えば、区間 $[0, \pi]$ 上定義された関数 f で、$f'(0) = f'(\pi) = 0$ を満たすものがあったとき、まず f を偶関数として区間 $[-\pi, \pi]$ まで拡張し、それから周期 2π の関数として $\mathbf{R}$ 全体に拡張した関数 $\tilde{f}$ を Fourier 級数展開し、それを $[0, \pi]$ に制限して f の表示式を得よう、とする場合など。$\tilde{f}$ の一周期区間 $[0, 2\pi]$ を奇数個の区間に分割すると、もとの関数 f の定義域 $[0, \pi]$ の端点である π は小区間の端点にならない。$[0, \pi]$ を等分するようなのがフツウに決まってる、でしょ?

とおけば、考えるべき写像は次のようになる。

定義 2.8 (離散余弦変換) 自然数 N_h を固定する。

$$A_k = \frac{1}{N_h}\left(f_0 + 2\sum_{j=1}^{N_h-1} f_j \cos\frac{\pi kj}{N_h} + (-1)^k f_{N_h}\right) \quad (k = 0, 1, \cdots, N_h) \tag{35}$$

で定められる写像

$$\mathbf{R}^{N_h+1} \ni (f_0, f_1, \cdots, f_{N_h}) \longmapsto (A_0, A_1, \cdots, A_{N_h}) \in \mathbf{R}^{N_h+1} \tag{36}$$

のことを**離散余弦変換**という。

(2016/12/3 記: 理由があって、久しぶりに見返したら、間違えていることを発見した。二つある N_h のうちの片方を $N_h - 1$ とするところ、取り違えをしてしまっていた。)

FFTPACK では、この問題を扱うために、手続き **costi**, **cost** が用意してある。(数列としての項数 $n \overset{\text{def.}}{=} N_h + 1$ を用いて記述してあるため、少し見ただけでは同じに見えないかもしれない。)

2.4 実数値奇関数に対する DFT — 離散正弦変換

ここでも 2.2 の記号を用いる。

関数 $f\colon S^1 \to \mathbf{R}$ が奇関数である、すなわち

$$f(-x) = -f(x) \quad (x \in S^1) \tag{37}$$

を満足すると仮定すると、Fourier 係数のうち cos の項に対応するものは 0 になる:

$$a_k = 0 \quad (k \in \mathbf{Z}). \tag{38}$$

やはり N 項実 DFT を考える。f が奇関数であることから、関数値の作る列 $\{f_j\}$ についても、

$$f_j = -f_{N-j} \quad (j \in \mathbf{Z}), \quad \text{よって } f_0 = 0. \text{ 特に } N \text{ が偶数の時 } f_{N/2} = 0 \tag{39}$$

が成り立つことに注意すると、K を

$$K = \begin{cases} N/2 - 1 & (N \text{ が偶数の場合}) \\ (N-1)/2 & (N \text{ が奇数の場合}) \end{cases} \tag{40}$$

で定めて、

$$(f_1, f_2, \cdots, f_K) \longmapsto (B_1, B_2, \cdots, B_K) \tag{41}$$

という写像を扱うと考えるのがよいことが分かる。(24), (25), (26), (27) から

$$B_k = \frac{2}{N}\sum_{j=0}^{N-1} f_j \sin\frac{2\pi kj}{N} = \frac{4}{N}\sum_{j=1}^{K} f_j \sin\frac{2\pi kj}{N} \quad (k = 1, 2, \cdots, K). \tag{42}$$

半分の添字 $j = 1, \cdots, K$ しか使わない理由は、「奇関数では、半周期分の関数値のデータを得れば、全体が再生出来る」ということで、理解しやすい。実際の応用では、半周期区間を等分するということで、ほとんどすべての場合に N は偶数である[13]。このとき

$$N_h \overset{\text{def.}}{=} \frac{N}{2} \tag{43}$$

[13] 2.3 の解説を参照のこと。

とおけば、考えるべき写像は次のようになる。

定義 2.9 (離散正弦変換) 自然数 N_h を固定する。

$$B_k = \frac{2}{N_h} \sum_{j=1}^{N_h-1} f_j \sin \frac{\pi kj}{N_h}. \quad (k = 1, \cdots, N_h - 1) \tag{44}$$

で定められる写像

$$\mathbf{R}^{N_h-1} \ni (f_1, f_2, \cdots, f_{N_h-1}) \longmapsto (B_1, B_2, \cdots, B_{N_h-1}) \in \mathbf{R}^{N_h-1} \tag{45}$$

のことを**離散正弦変換**という。

FFTPACK では、この問題を扱うために手続き **sinti**, **sint** を用意している。(数列の項数 $n \overset{\text{def.}}{=} N_h - 1$ を用いて記述してあるため、少し見ただけでは同じに見えないかもしれない。)

3 FFTPACK の紹介

3.1 FFTPACK の生い立ち

FFT は非常に重要なので、多くの書籍にプログラムが掲載されているが、残念なことに、それらの多くは、項数 N が 2 の冪である、すなわち

$$\exists m \in \mathbf{N} \quad \text{s.t.} \quad N = 2^m \tag{46}$$

である場合にしか使えない。大抵の応用には、(46) であるように N を取ることは簡単なので、それで十分なのだが、(46) であるように N を取るのが難しい場合もある。

FFT の原理そのものは、N が小さな素数のみを素因数に持つ場合に拡張できるので、それ — Winograd の Fourier 変換とも呼ばれる — を実現したプログラムも存在する。

ここでは Paul N. Swarztrauber による FFTPACK を紹介する。これはいわゆる public domain software で、ソースも公開されていて、自由に使用することが出来る[14]。コピーライトと内容一覧を以下に載せておく。

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

                  version 4  april 1985

    a package of fortran subprograms for the fast fourier
     transform of periodic and other symmetric sequences

                         by

                  paul n swarztrauber

  national center for atmospheric research  boulder,colorado 80307

   which is sponsored by the national science foundation
```

[14]商用数値計算ライブラリィとして有名な IMSL にも使われている、とのことである。

* *

このパッケージは、以下に列挙するような、実および複素周期数列やある種の対称数列に対する高速フーリエ変換のプログラムから構成されている。

rffti rfftf と rfftb を初期化する

rfftf 実周期数列の順変換

rfftb 実係数配列の逆変換

ezffti ezfftf と ezfftb を初期化する

ezfftf 単純化された実周期の順変換

ezfftb 単純化された実周期の逆変換

sinti sint を初期化する

sint 実奇関数列のサイン変換

costi cost を初期化する

cost 実偶関数列のコサイン変換

sinqi sinqf と sinqb を初期化する

sinqf 奇波数の順サイン変換

sinqb sinqf の非正規化逆変換

cosqi cosqf と cosqb の初期化

cosqf 奇波数の順コサイン変換

cosqb cosqf の非正規化逆変換

cffti cfftf と cfftb の初期化

cfftf 複素周期数列の順変換

cfftb 複素周期数列の非正規化逆変換

3.2 FFTPACK に用意されている手続きの使い方

ここでは FFTPACK に用意されている手続きの使い方を説明する。

手続きの使い方を理解するためには、サンプル・プログラムを読むのが役に立つ。以下では熱方程式の初期値・境界値問題から例題を作り、FFTPACK を利用して解くプログラムを示す。

3.2.1 サンプル・プログラムについて

この文書は、明治大学数学科のワークステーション環境下で利用することを念頭においてあるため、以下のサンプル・プログラムの中には、FORTRAN や C の組み込み関数・サブルーチンでないものも一部使っている。ここでは、それらについて簡単に解説する。

getint(), getdouble()　C 言語のプログラムで、標準入力から数値を読み込むためには、例えば **scanf()** を使うことも出来るが、これはある種の不正な入力に対して、プロセスを一切回復不能なドツボ状況に陥らせ得る、ということで、使用するのはあまり望ましくない、と良く言われている[15]。そこで以下のプログラムでは、整数を読み込むために **getint()**、倍精度実数を読み込むために **getdouble()** という関数を用いている。これは、例えば次のようにコーディングしたものである。

```
/*
 * get_id.c -- definition of getint(), getdouble()
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int getint()
{
    char buf[512];
    if (fgets(buf, sizeof(buf), stdin) == NULL) {
        fprintf(stderr, "getint: Unexpected EOF\n");
        return 0;
    }
    return atoi(buf);
}

double getdouble()
{
    char buf[512];
    if (fgets(buf, sizeof(buf), stdin) == NULL) {
        fprintf(stderr, "getdouble: Unexpected EOF\n");
        return 0.0;
    }
    return atof(buf);
}
```

反省　と二年前は言ったのだけれど、演習用のプログラムでこんなことをするのはやはり変 (神経質過ぎる) かも知れない、と今は思う (下の脚注に書いた気持ちが強くなってきた)。`getint()`, `getdouble()` を打ち込むのが面倒に感じたら、

```
n = getint();
...
x = getdouble();
```

を

```
scanf("%d", &n);
...
scanf("%lf", &x);
```

のように直す要領で書き換えをして下さい。

[15] とはいえ、“garbage in, garbage out” — 「ゴミを入れれば、ゴミしか出て来ない」 — は世の習いで、どうせ入力を間違ったのならば仕方がない (最初からやり直そう)、という考え方も、少なくとも数値シミュレーション用のプログラムではあり得る、とも筆者は思う。システム・プログラムでは絶対に許されないことだろうが。

グラフィッスク用の命令　以下のプログラムでは、グラフを描くために、**openpl()**, **closepl()**, **fspace2()**, **linemod()**, **fline()**, **fmove()**, **fcont()**, **mkplot()** のような関数を用いている。これは UNIX のオンライン・マニュアルの第 3 章にある一連の plot ライブラリィ関数の浮動小数点数バージョンである (**mkplot()** のみはオリジナルで、これは描画した図をファイルに保存する命令である)。UNIX 環境下で "`man 3 plot`" としてマニュアルを読めば、その内容は推測が可能で、別のグラフィックス・ライブラリィを用いるようにプログラムを書き換えることは簡単であろう。

3.2.2　複素 FFT

複素 DFT は

cffti 以下の **cfftf**, **cfftb** で用いる定数表を初期化する。

cfftf 順方向 DFT、すなわち関数値の列 $\{f_j\}$ から近似 Fourier 係数 $\{\widehat{f}^a(k)\}$ を求める

cfftb 逆方向 DFT、すなわち近似 Fourier 係数 $\{\widehat{f}^a(k)\}$ から関数値の列 $\{f_j\}$ を求める

の三つの手続きでサポートされている。数列の長さ N が小さな素数のみを素因数に持つ場合は、FFT のアルゴリズムが使われて、極めて効率的に実行される。

これらを利用するには、数列を格納するための配列を

```
complex c(MAXN)                    complex c[MAXN];
```

のように宣言する必要があるのはもちろんであるが、サブルーチンが計算に必要とする作業領域も

```
real wsave(4*MAXN+15)              double wsave[4*MAXN+15];
```

のように宣言して、**cfftf**, **cfftb** の呼び出しに先立って、

```
call cffti(N, wsave)               cffti(N, wsave);
```

のように **cffti** を呼び出して、初期化しておく必要がある。こうすると、この $wsave$ には、数列の長さ N に対応した定数表 (N の素因数分解の結果、必要な三角関数の値を含む) が格納されることになる。

Fourier 係数を求めるには、まず配列 c() に係数を求めようと思う複素数値関数 f の、N 等分点上での値を書き込む必要がある。これには、例えば

```
h = 2.0 * PI / N                   h = 2.0 * PI / N;
do j=1,N                           for (j = 0; j < N; j++)
  c(j) = f((j-1)*h)                  c[j] = f(j * h);
end do
```

のようなコーディングをすればよい (もちろん `PI` は円周率 π である)。こうしておいてから、

```
call cfftf(N, c, wsave)            cfftf(N, c, wsave);
```

のように **cfftf** を呼び出すと、配列 c() は f の Fourier 係数の近似値の N 倍 $N\widehat{f}^a(0)$, $N\widehat{f}^a(1)$, $\cdots$, $N\widehat{f}^a(N-1)$ で上書きされる。Fourier 係数の近似値そのものが欲しければ、

```
do k=1,N                          for (k = 0; k < N; k++)
  c(k) = c(k) / N                   c[k] /= N;
end do
```

のように自分で割算をする必要がある。なお、この **cfftf** 呼び出しで $wsave$ は書き換えられないので、以下の (同じ N に対する) **cfftf**, **cfftb** の呼び出しの際には、**cffti** を再び呼び出す必要はない。

反対に、Fourier 係数から、関数の値の列を求めるには、

```
call cfftb(N, c, wsave)           cfftb(N, c, wsave);
```

と —cfftb を呼び出せばよい。

以上の説明で、FORTRAN は単精度のプログラムを書くとして説明したが、倍精度にするには、

```
real wsave(4*MAXN+1)  →  real*8 wsave(4*MAXN+1)
complex c(MAXN)       →  complex*16 c(MAXN)
```

のような書換えが必要である。一方で、添字を 0 から番号づけることも出来る。実行文に関しては C プログラムを参考にすればいいが、宣言に関しては、

```
complex c(MAXN) ⟶ complex c(0:MAXN-1)
```

のように書き換える。

複数の N の値に対して DFT を行う場合は、各々の N に対応した表を作るために **cffti** を呼び出す必要があるが、この場合も元の $wsave$ を保存しておいて、後で再び使うことが可能である。例えば

```
call cffti(N1, wsave1)
call cffti(N2, wsave2)
....
call cfftf(N1, r1, wsave1)
call cfftb(N2, r2, wsave2)
call cfftf(N1, r3, wsave1)
```

のように、$N1$, $N2$ に対応する表をそれぞれ $wsave1$, $wsave2$ に入れておいて、必要に応じてスイッチして使うようにも出来る。

3.2.3 実数版 DFT

実数版 DFT を扱うのに、二系統の手続きの組を用意している。まず、手続き名の先頭が “ezfft” である三つの手続きの組、

ezffti 以下の **ezfftf**, **ezfftb** で用いる関数値表の初期化をする

ezfftf 順方向 DFT、すなわち関数値の列 $\{f_j\}$ から近似 Fourier 係数 $\{A_k\}$, $\{B_k\}$ を求める

ezffti 逆方向 DFT、すなわち近似 Fourier 係数 $\{A_k\}$, $\{B_k\}$ から関数値の列 $\{f_j\}$ を求める

は扱い方の簡単な手続きとして、気軽に FFTPACK を利用しようという人にお勧めできる。しかし、効率について厳しく考えると、若干の無駄があることは否めない。極力無駄を省きたいという人には、手続き名の先頭が “rfft” である三つの手続きの組、

rffti 次の **rfftf**, **rfftb** で用いる関数値表の初期化をする

rfftf 順方向 DFT、すなわち関数値の列 $\{f_j\}$ から近似 Fourier 係数 $\{A_k\}$, $\{B_k\}$ を求める

rffti 逆方向 DFT、すなわち近似 Fourier 係数 $\{A_k\}$, $\{B_k\}$ から関数値の列 $\{f_j\}$ を求める

を使うことをすすめる。

それでは、**ezfft{i,f,b}** から説明しよう。**ezffti** は **ezfftf**, **ezfftb** を利用するために必要となる定数表を初期化するためのもので、

```
real wsave(3*MAXN+15)                    double wsave[3*MAXN+15];
```

のように作業用配列 $wsave$ を宣言しておいてから、**ezfftf**, **ezfftb** の呼び出しに先立って、

```
call ezfffi(N, wsave)                    ezfffi(N, wsave);
```

のように呼び出して、$wsave$ を初期化する必要がある。こうすると、この $wsave$ には、数列の長さ N に対応した定数表 (N の素因数分解の結果、必要な三角関数の値を含む) が格納されることになる。

さて、作業用配列以外には、関数値を納める配列 r() と、Fourier 係数を納める配列 a(), b(), さらには Fourier 係数 A_0 を納める変数 $azero$ が必要になる。これらは

```
real r(MAXN),a(MAXN/2),b(MAXN/2),azero   double r[MAXN], a[MAXN/2+1],
                                          b[MAXN/2+1];
```

のように宣言しておく。

Fourier 係数を求めるには、まず配列 r() に係数を求めようと思う実数値関数 f の、N 等分点上での値を書き込む必要がある。これには、例えば

```
h = 2.0 * PI / N                         h = 2.0 * PI / N;
do j=1,N                                 for (j = 0; j < N; j++)
  r(j) = f((j-1)*h)                        r[j] = f(j * h);
end do
```

のようなコーディングをすればよい (もちろん PI は円周率 π である)。こうしておいてから、

```
call ezfftf(N,r,azero,a,b,wsave)         ezfftf(N, r, &a[0], a+1, b+1,
                                         wsave);
```

のように **ezfftf** を呼び出すと、

$azero$	$=$	$A_0/2$	a[0]	$=$	$A_0/2$
a(k)	$=$	$A_{\mathtt{k}}$ (k$=1,2,\cdots,L$)	a[k]	$=$	$A_{\mathtt{k}}$ (k$=1,2,\cdots,L$)
b(k)	$=$	$B_{\mathtt{k}}$ (k$=1,2,\cdots,L$)	b[k]	$=$	$B_{\mathtt{k}}$ (k$=1,2,\cdots,L$)

となる。ただし L は

$$L = \begin{cases} N/2 & (N \text{ が偶数}) \\ (N-1)/2 & (N \text{ が奇数}) \end{cases}$$

で定義される自然数である。ここで、$k=0$ だけ特別扱いになっていることに関して、少し説明をしておく。FFTPACK の開発者である P. N. Swartzrauber は、Fourier 級数を (15) でなく、

$$f(x) = a_0 + \sum_{k=1}^{\infty}(a_k \cos kx + b_k \sin kx) \tag{47}$$

とみなしている、すなわち Fourier 係数の定義を (16) ではなく、

$$\begin{aligned} a_0 &= \frac{1}{2\pi}\int_0^{2\pi} f(x)\,dx \\ a_k &= \frac{1}{\pi}\int_0^{2\pi} f(x)\cos kx\,dx \quad (k=1,2,\cdots) \\ b_k &= \frac{1}{\pi}\int_0^{2\pi} f(x)\sin kx\,dx \quad (k=1,2,\cdots). \end{aligned} \tag{48}$$

と考えているらしい。$k=1,2,\cdots$ に対する a_k, b_k は我々の流儀と同じだが、a_0 が異なる。つまり

$$\text{FFTPACK の } a_0 = \text{我々の } a_0 \text{ の } 1/2. \tag{49}$$

注意を要するのは、FFTPACK それ自身は首尾一関していて、例えば次の **ezfftb** は (正規化されてはいないものの) **ezfftf** のきれいな逆になっているので、*azero* (C 言語の場合は `a[0]`) を勝手に 2 倍すると、後で **ezfftb** を呼び出す時に、2 で割る必要が出て来る可能性がある、ということである。ここは、素直に FFTPACK の流儀に則ってプログラムを書くことを勧める。

Fourier 係数から関数値の列を計算するには **ezfftb** を

`call` **ezfftb**`(`N`,`r`,`*azero*`,`a`,`b`,`*wsave*`)`	**ezfftb**`(`N`, `r`, &`a`[0], `a`+1, `b`+1, `*wsave*`);`

のように呼び出す。ここでは (**ezfft** とは逆で) N, `a(k)`, `b(k)` (k$=1,2,\cdots,L$), *azero* (C の場合は `a[0]`) が入力で、`r()` が出力である。N が偶数の場合、`b(N/2)` を 0 にセットしておくこと[16]。

ezfftf, **ezfftb** の計算の内容は、A_0 の問題を別にすれば、2.2 で説明した実 DFT そのものである。

最後に、Fourier 係数全体を操作するループをコーディングするには、どうすればよいかについて一言。2.2 では、N が偶数のときは $A_0, A_1, B_1, A_2, B_2, \cdots, A_{N/2-1}, B_{N/2-1}$ に意味があり、N が奇数のときは $A_0, A_1, B_1, A_2, B_2, \cdots, A_{(N-1)/2}, B_{(N-1)/2}$ に意味がある、と書いたが、

- **ezfftf** では、N が偶数のとき `b(N/2)` も出力する (内容は 0)。
- FORTRAN や C で N が整数型の変数のとき、値が正の奇数であれば `N/2` の値は $(N-1)/2$ に等しい。

等に注意すれば、

azero に対する処理	L `= N / 2;` `a[0]` に対する処理
`do k=1,`N`/2`	`for (k = 1; k <=` L`; k++) {`
`a(k), b(k)` に対する処理	`a[k], b[k]` に対する処理
`end do`	`}`

とすればよいことが分かる。

次に **rfft{i,f,b}** を説明しよう。**rffti** は **rfftf**, **rfftb** を利用するために必要となる定数表を初期化するためのもので、

[16] 本当に必要なのかな？

```
real wsave(2*MAXN+15)
```
```
double wsave[2*MAXN+15];
```

のように作業用配列 $wsave$ を宣言しておいてから、**rfftf**, **rfftb** の呼び出しに先立って、

```
call rffti(N, wsave)
```
```
rffti(N, wsave);
```

のように呼び出して、$wsave$ を初期化する必要がある。こうすると、この $wsave$ には、数列の長さ N に対応した定数表 (N の素因数分解の結果、必要な三角関数の値を含む) が格納されることになる。

さて、作業用配列以外には、関数値と Fourier 係数兼用の配列 r() が必要になる。これらは

```
real r(MAXN)
```
```
double r[MAXN];
```

のように宣言しておく。

Fourier 係数を求めるには、まず配列 r() に係数を求めようと思う実数値関数 f の、N 等分点上での値を書き込む必要がある。これには、例えば

```
h = 2.0 * PI / N
do j=1,N
  r(j) = f((j-1)*h)
end do
```
```
h = 2.0 * PI / N;
for (j = 0; j < N; j++)
  r[j] = f(j * h);
```

のようなコーディングをすればよい (もちろん PI は円周率 π である)。こうしておいてから、

```
call rfftf(N,r,wsave)
```
```
rfftf(N, r, wsave);
```

のように **rfftf** を呼び出すと、配列 r() に、順に

$$\frac{N}{2}A_0, \frac{N}{2}A_1, -\frac{N}{2}B_1, \frac{N}{2}A_2, -\frac{N}{2}B_2, \cdots, \frac{N}{2}A_L, -\frac{N}{2}B_L$$

が格納される。つまり **rfftf** は 2.2 で説明した順方向実 DFT の $N/2$ 倍を計算する (B_k については符号が異なるが)。実 DFT そのものを計算したい場合は、 **rfftf** の呼び出しの後に[17]

```
do k=1,N
  r(k) = r(k)/NOVER2
end do
```
```
for (k = 0; k < N; k++)
  r[k] /= NOVER2;
```

のように割算をするわけである。ここで NOVER2 はもちろん $N/2$ を値にもつ (これは整数の割算で計算してはいけないので、NOVER2 は実数型として宣言して、 NOVER2 = N / 2.0 または NOVER2 = 0.5 * N のように計算する)。

この逆に Fourier 係数から関数値の列を求めるには **cfftb** を用いる。これは 2.2 で説明した逆方向実 DFT の 2 倍を計算する。呼び出し方は

```
call rfftb(N,r,wsave)
```
```
rfftb(N, r, wsave);
```

である。きまじめに Fourier 解析に続いて、Fourier 合成を行なうには

```
call rfftf(N,r,wsave)
do k=1,N
  r(k) = r(k)/NOVER2
end do
何かの処理
call rfftb(N,r,wsave)
do j=1,N
  r(j) = r(j)/2.0
end do
```
```
rfftf(N, r, wsave);
for (k = 0; k < N; k++)
  r[k] /= NOVER2;
何かの処理
rfftb(N, r, wsave);
for (j = 0; j < N; j++)
  r[j] /= 2.0;
```

[17]別に前にやっても構わない。

のようなプログラムとなるが、この「何かの処理」の中で、配列 r() に対して行なわれる操作が、線形演算であるならば、二つの割算を一つの割算 (N で割る) にまとめること、例えば

```
call rfftf(N,r,wsave)
do k=1,N
  r(k) = r(k)/N
end do
何かの処理
call rfftb(N,r,wsave)
```

```
rfftf(N, r, wsave);
for (k = 0; k < N; k++)
  r[k] /= N;
何かの処理
rfftb(N, r, wsave);
```

のように出来る。

例題: 次の初期値・境界値問題を解くプログラムを作れ。

$$
(50)\qquad \begin{cases} u_t(x,t) &= u_{xx}(x,t) & (x\in(0,2\pi),\ t>0)\\ u(0,t) &= u(2\pi,t),\quad u_x(0,t)=u_x(2\pi,t) & (t>0)\\ u(x,0) &= f(x) & (x\in[0,2\pi]). \end{cases}
$$

ただし $f(x)=1+2\cos x+3\sin 2x+4\cos 3x+5\sin 3x$.

解答: 問題は S^1 上の方程式である。初期データ f を

$$
(51)\qquad f(x)=\frac{a_0}{2}+\sum_{k=1}^{\infty}(a_k\cos kx+b_k\sin kx)
$$

のように Fourier 級数展開したとすると、

$$
(52)\qquad u(x,t)=\frac{a_0}{2}+\sum_{k=1}^{\infty}e^{-k^2t}(a_k\cos kx+b_k\sin kx)
$$

で解が求まる。これを **ezfft** を使って解くプログラムを紹介しよう。

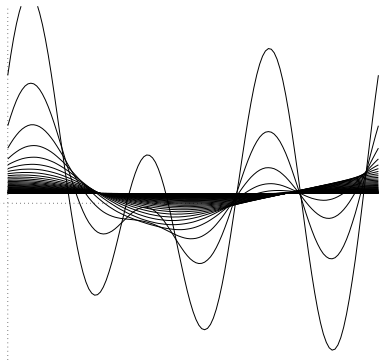
```
/*
 * tstezfft.c -- test of ezffti(), ezfftf(), ezfftb() in FFTPACK
 *   How to compile: ccmg tstezfft.c get_id.c -ldfftpack
 */

#include <stdio.h>
#include <math.h>
#include <glsc.h>
#include "dfftpack.h"

#define MAXN    (4096)
#define HMAXN   (MAXN/2)

double f(double);

int main(void)
{
    /* ユーザーが入力する値 */
    int N;
    double Tmax, tau;
    /* 定数 */
    double PI;
    /* プログラムの実行を通じて不変な値を持つ変数，関数宣言 */
```

```
int halfN, maxnstep, getint();
double h, f(), getdouble();
/* 制御変数, 作業用配列 */
int j, k, nstep;
double t;
double r[MAXN+1], a[HMAXN+1], b[HMAXN+1], work[3*MAXN+15];
double at[HMAXN+1], bt[HMAXN+1];

/* 定数値の設定 */
PI = 4.0 * atan(1.0);
/* 分割数 N, 追跡時間 Tmax, 時間刻み幅 τ の決定 */
printf("N=");    N    = getint();    if (N <= 0 || N > MAXN) return 0;
printf("Tmax="); Tmax = getdouble();
printf("τ=");   tau  = getdouble(); if (tau == 0.0) return 0;
/* プログラムの実行を通じて不変な値を持つ変数 */
halfN = N / 2; maxnstep = Tmax / tau; h = 2.0 * PI / N;
if (maxnstep < 0) {
    fprintf(stderr, "Tmax, τの値を見直して下さい\n");
    return 0;
}
/* 初期値 */
for (j = 0; j < N; j++) r[j] = f(j * h);
/* N 項実 FFT の準備の後、順方向 DFT によって Fourier 係数を求める */
ezffti(N, work);
ezfftf(N, r, &a[0], a+1, b+1, work);
/* Fourier 係数の最初の数項を表示 */
printf("a[0]/2=%15f\n", a[0]);
for (k = 1; k < 5; k++)
    printf("a[%d]  =%15f, b[%d]  =%15f\n", k, a[k], k, b[k]);
/* ウィンドウを開き、適当に座標を入れる */
g_init("EZFFT", 140.0, 140.0);
g_device(G_BOTH);
g_def_scale(0,
            -0.2, 6.48, -10.0, 10.0,
            10.0, 10.0, 120.0, 120.0);
g_sel_scale(0);
/* 座標軸を描く */
g_def_line(0, G_BLACK, 2, G_LINE_DOTS);
g_def_line(1, G_BLACK, 1, G_LINE_SOLID);
g_sel_line(0);
g_move(-0.2, 0.0); g_plot(6.48, 0.0);
g_move(0.0, -10.0); g_plot(0.0, 10.0);
g_sel_line(1);
/* */
for (nstep = 0; nstep <= maxnstep; nstep++) {
    t = nstep * tau;
    /* u(・,t) の Fourier 係数を求める */
    at[0] = a[0];
    for (k = 1; k <= halfN; k++) {
        double dmpfctr = exp(- k * k * t);
        at[k] = a[k] * dmpfctr;
        bt[k] = b[k] * dmpfctr;
    }
    /* 逆実 DFT により、関数値を求める */
    ezfftb(N, r, &at[0], at+1, bt+1, work);
    /* グラフを描く */
```

```
        r[N] = r[0];
        g_move(0.0, r[0]);
        for (j = 1; j <= N; j++) g_plot(j * h, r[j]);
    }
    g_sleep(G_STOP);
    return 0;
}

double f(double x)
{
    /* a0/2+(a1 cos(x)+b1 sin(x))+...+(a3 cos(3x)+b3 sin(3x)) */
    int i;
    static double a[4] = {1.0, 2.0, 0.0, 4.0};
    static double b[4] = {0.0, 0.0, 3.0, 5.0};
    double result = a[0] / 2.0;
    for (i = 1; i < 4; i++)
        result += a[i] * cos(i * x) + b[i] * sin(i * x);
    return result;
}
```

これを実行させると以下のようにな結果になる。

```
oyabun% ccx tstezfft.c get_id.o -ldfftpack
oyabun% tstezfft
N=128
Tmax=5.0
B=0.1
a[0]/2=       0.500000
a[1]  =       2.000000, b[1]  =       0.000000
a[2]  =      -0.000000, b[2]  =       3.000000
a[3]  =       4.000000, b[3]  =       5.000000
a[4]  =       0.000000, b[4]  =       0.000000
oyabun%
```



もちろん、**rfft** を使うことも出来る。

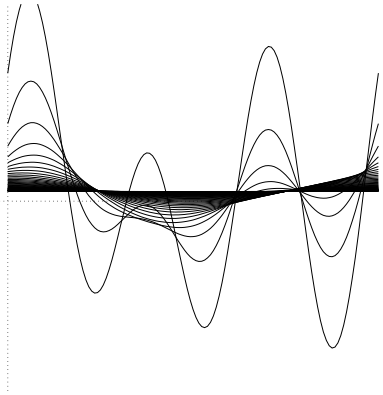
```
/*
 * tstrfft.c -- test of rffti(), rfftf(), rfftb() in FFTPACK
 *   How to compile: ccmg tstrfft.c get_id.o -ldfftpack
 */

#include <stdio.h>
#include <math.h>
#include <glsc.h>
#include "dfftpack.h"
```

```
#define MAXN    (4096)

double f(double);

int main(void)
{
    /* ユーザーが入力する値 */
    int N;
    double Tmax, tau;
    /* 定数 */
    double PI;
    /* プログラムの実行を通じて不変な値を持つ変数，関数宣言 */
    int maxnstep, getint();
    double h, f(), getdouble();
    /* 制御変数，作業用配列 */
    int i, j, nstep;
    double t;
    double r[MAXN], rt[MAXN+1], work[2*MAXN+15];

    /* 定数値の設定 */
    PI = 4.0 * atan(1.0);
    /* 分割数 N，追跡時間 Tmax，時間刻み幅 τ の決定 */
    printf("N=");    N    = getint();    if (N <= 0 || N > MAXN) return 0;
    printf("Tmax="); Tmax = getdouble();
    printf("τ=");   tau  = getdouble(); if (tau == 0.0) return 0;
    /* プログラムの実行を通じて不変な値を持つ変数 */
    maxnstep = Tmax / tau; h = 2.0 * PI / N;
    if (maxnstep < 0) {
        fprintf(stderr, "Tmax, τの値を見直して下さい\n");
        return 0;
    }
    /* 初期値 */
    for (j = 0; j < N; j++) r[j] = f(j * h);
    /* N 項実 FFT の準備の後、順方向 DFT によって Fourier 係数を求める
     * よくある話だが、rfftf() で実際に計算されるのは、本来の Fourier
     * 係数を N/2 倍したものである。一方、後の rfftb で計算されるのは
     * 逆 DFT の 2 倍である。そこで rfftf() に引き続いて rfftb() を実
     * 行すると数列を N 倍することになる。ここでは rfftf() を呼んだ後に、
     * 全体を N で割っている。こうすることにより、しばらく r[] には、
     * 本来の Fourier 係数の 1/2 倍が収められることになるが、全体とし
     * ては割り算の回数を節約できる。 */
    rffti(N, work);
    rfftf(N, r, work);
    for (i = 0; i < N; i++) r[i] /= N;
    /* Fourier 係数の最初の数項を表示 */
    printf("r[0]/2=%15f\n", r[0]);
    for (i = 1; i < 8; i += 2)
        printf("r[%d]/2=%15f, r[%d]/2=%15f\n", i, r[i], i+1, r[i+1]);
    /* ウィンドウを開き、適当に座標を入れる */
    g_init("RFFT", 140.0, 140.0);
    g_device(G_BOTH);
    g_def_scale(0,
                -0.2, 6.48, -10.0, 10.0,
                10.0, 10.0, 120.0, 120.0);
    g_sel_scale(0);
```

```
    /* 座標軸を描く */
    g_def_line(0, G_BLACK, 2, G_LINE_DOTS);
    g_def_line(1, G_BLACK, 1, G_LINE_SOLID);
    g_sel_line(0);
    g_move(-0.2, 0.0); g_plot(6.48, 0.0);
    g_move(0.0, -10.0); g_plot(0.0, 10.0);
    g_sel_line(1);
    /* */
    for (nstep = 0; nstep <= maxnstep; nstep++) {
        t = nstep * tau;
        /* u(・,t) の Fourier 係数を求める */
        rt[0] = r[0];
        for (i = 1; i < N; i += 2) {
            int k = (i + 1) >> 1; /* k = (i + 1) / 2; */
            double dmpfctr = exp(- k * k * t);
            rt[  i] = r[  i] * dmpfctr;
            rt[i+1] = r[i+1] * dmpfctr;
        }
        /* 逆実 DFT により、関数値を求める */
        rfftb(N, rt, work);
        /* グラフを描く */
        rt[N] = rt[0];
        g_move(0.0, rt[0]);
        for (j = 1; j <= N; j++) g_plot(j * h, rt[j]);
    }
    g_sleep(G_STOP);
    return 0;
}

double f(double x)
{
    /* a0/2+(a1 cos(x)+b1 sin(x))+...+(a3 cos(3x)+b3 sin(3x)) */
    int i;
    static double a[4] = {1.0, 2.0, 0.0, 4.0};
    static double b[4] = {0.0, 0.0, 3.0, 5.0};
    double result = a[0] / 2.0;
    for (i = 1; i < 4; i++)
        result += a[i] * cos(i * x) + b[i] * sin(i * x);
    return result;
}
```

これを実行させると以下のようにな結果になる。

```
oyabun% ccx tstrfft.c get_id.o -ldfftpack
oyabun% tstrfft
N=128
Tmax=5.0
Ｂ=0.1
r[0]/2=       0.500000
r[1]/2=       1.000000, r[2]/2=      -0.000000
r[3]/2=      -0.000000, r[4]/2=      -1.500000
r[5]/2=       2.000000, r[6]/2=      -2.500000
r[7]/2=       0.000000, r[8]/2=      -0.000000
oyabun%
```



3.2.4 実数値偶関数に対する DFT — 余弦変換

実数値偶関数に対する DFT — 余弦変換 —、のための手続きとして、

costi 次の **cost** で用いる定数表を初期化する。

cost $\{f_j\} \longmapsto \{A_k\}$ を計算する。その逆 $\{A_k\} \longmapsto \{f_j\}$ もこれで計算できる (後で正規化が必要になる)。

が用意されている。

costi は **cost** を利用するために必要となる定数表を初期化するためのもので、

```
real wsave(3*MAXn+15)                    double wsave[3*MAXn+15];
```

のように作業用配列 $wsave$ を宣言しておいてから、**cost** の呼び出しに先立って、

```
call costi(n, wsave)                     costi(n, wsave);
あるいは                                  あるいは
call costi(Nh+1, wsave)                  costi(Nh+1, wsave);
```

のように呼び出して、$wsave$ を初期化する必要がある。こうすると、この $wsave$ には、n に対応した定数表が格納されることになる。ここで n は **cost** の引数となる数列の長さで N_h+1 に等しい。

さて、作業用配列以外には、関数値と Fourier 係数兼用の配列 x() が必要になる。これらは

```
real x(MAXn) あるいは real x(MAXNh+1)   double x[MAXn]; あるいは double x[MAXNh+1];
```

のように宣言しておく。

Fourier 係数を求めるには、まず配列 x() に係数を求めようと思う実数値偶関数 f の、半周期区間 $[0,\pi]$ の N_h 等分点上での値を書き込む必要がある。これには、例えば

```
h = PI / Nh                              h = PI / Nh;
do j=1,Nh+1                              for (j = 0; j <= Nh; j++)
  x(j) = f((j-1)*h)                        x[j] = f(j * h);
end do
```

のようなコーディングをすればよい (もちろん PI は円周率 π である)。こうしておいてから、

```
call cost(n,x,wsave)                     cost(n, x, wsave);
あるいは                                  あるいは
call cost(Nh+1,x,wsave)                  cost(Nh+1, x, wsave);
```

のように **cost** を呼び出すと、配列 $x()$ に、順に

$$N_h A_0, N_h A_1, N_h A_2, \cdots, N_h A_{N_h}$$

$N_h + 1$ 個の数が格納される。つまり **cost** は 2.3 で説明した順方向実 DFT の N_h 倍を計算する。実 DFT そのものを計算したい場合は、 **cost** の呼び出しの後に[18]

```
do k=1,Nh+1
  x(k) = x(k)/Nh
end do
```

```
for (k = 0; k <= Nh; k++)
  x[k] /= Nh;
```

のように割算をするわけである。

この逆に Fourier 係数から関数値の列を求めるにも **cost** を用いる。これは 2.3 で説明した逆方向離散余弦変換 の 2 倍を計算する。

きまじめに Fourier 解析に続いて、Fourier 合成を行なうには

```
call cost(Nh+1,x,wsave)
do k=1,Nh+1
  x(k) = x(k)/Nh
end do
何かの処理
call cost(Nh+1,x,wsave)
do j=1,Nh+1
  x(j) = x(j)/2.0
end do
```

```
cost(Nh+1, x, wsave);
for (k = 0; k <= Nh; k++)
  x[k] /= Nh;
何かの処理
cost(Nh+1, x, wsave);
for (j = 0; j <= Nh; j++)
  x[j] /= 2.0;
```

のようなプログラムとなるが、この「何かの処理」の中で、配列 $x()$ に対して行なわれる操作が、線形演算であるならば、二つの割算を一つの割算 ($2N_h$ で割る) にまとめること、例えば

```
call cost(Nh+1,x,wsave)
do k=1,Nh+1
  x(k) = x(k)/twoNh
end do
何かの処理
call cost(Nh+1,x,wsave)
```

```
cost(Nh, x, wsave);
for (k = 0; k <= N+h; k++)
  x[k] /= twoNh;
何かの処理
cosb(Nh+1, x, wsave);
```

のように出来る。もちろん $twoN_h$ は $2N_h$ という値をもつ変数である。

例題: 次の初期値・境界値問題を解くプログラムを作れ。

$$\text{(53)} \qquad \begin{cases} u_t(x,t) &=& u_{xx}(x,t) & (x \in (0,\pi),\ t > 0) \\ u_x(0,t) &=& u_x(\pi,t) = 0 & (t > 0) \\ u(x,0) &=& f(x) & (x \in [0,\pi]). \end{cases}$$

ただし $f(x) = 1 + 2\cos x - 3\cos 2x + 4\cos 3x.$

解答: 初期データ f を

$$\text{(54)} \qquad f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty} a_k \cos kx$$

のように Fourier 級数展開したとすると、

$$\text{(55)} \qquad u(x,t) = \frac{a_0}{2} + \sum_{k=1}^{\infty} e^{-k^2 t} a_k \cos kx$$

で解が求まる。これを **cost** を使って解くプログラムを紹介しよう。

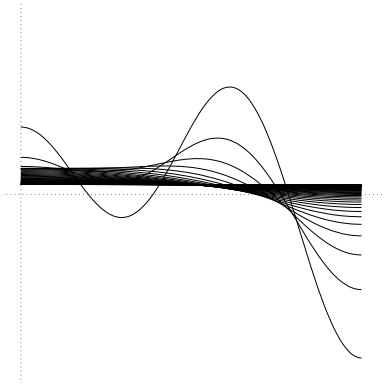
[18] 別に前にやっても構わない。

```
/*
 * tstcost.c -- test of costi(), cost() in FFTPACK
 *   How to compile: ccmg tstcost.c get_id.c -ldfftpack
 */

#include <stdio.h>
#include <math.h>
#include <glsc.h>
#include "dfftpack.h"

#define MAXN    (4096)

double f(double);

int main(void)
{
    /* ユーザーが入力する値 */
    int N;
    double Tmax, tau;
    /* 定数 */
    double PI;
    /* プログラムの実行を通じて不変な値を持つ変数，関数宣言 */
    int twoN, maxnstep, getint();
    double h, f(), getdouble();
    /* 制御変数，作業用配列 */
    int j, k, nstep;
    double t;
    double a[MAXN+1], work[3*MAXN+15], at[MAXN+1];

    /* 定数値の設定 */
    PI = 4.0 * atan(1.0);
    /* 分割数 N，追跡時間 Tmax，時間刻み幅 τ の決定 */
    printf("N=");    N    = getint();    if (N <= 0 || N > MAXN) return 0;
    printf("Tmax="); Tmax = getdouble();
    printf("τ=");   tau  = getdouble(); if (tau == 0.0) return 0;
    /* プログラムの実行を通じて不変な値を持つ変数 */
    twoN = 2 * N; maxnstep = Tmax / tau; h = PI / N;
    if (maxnstep < 0) {
        fprintf(stderr, "Tmax, τの値を見直して下さい\n");
        return 0;
    }
    /* 初期値 */
    for (j = 0; j <= N; j++) a[j] = f(j * h);
    /* N 項実 FFT の準備の後、順方向 DFT によって Fourier 係数を求める
     * costi(),cost() の第一引数は、数列の長さであり、区間の分割数では
     * ないことに注意しよう。だから N ではなくて、N+1 を指定する。 */
    costi(N+1, work);
    cost(N+1, a, work);
    /* cost() の出力を N でなく 2N で割っている。
     *  → 本当の Fourier 係数の 1/2 を得る。 */
    for (k = 0; k <= N; k++) a[k] /= twoN;
    /* Fourier 係数 (* 1/2) の最初の数項を表示 */
    for (k = 0; k < 5; k++)
        printf("a[%d]/2=%15f\n", k, a[k]);
    /* ウィンドウを開き、適当に座標を入れる */
    g_init("COST", 140.0, 140.0);
```

```
    g_device(G_BOTH);
    g_def_scale(0,
                -0.2, 3.35, -10.0, 10.0,
                10.0, 10.0, 120.0, 120.0);
    g_sel_scale(0);
    /* 座標軸を描く */
    g_def_line(0, G_BLACK, 2, G_LINE_DOTS);
    g_def_line(1, G_BLACK, 1, G_LINE_SOLID);
    g_sel_line(0);
    g_move(-0.2, 0.0); g_plot(3.35, 0.0);
    g_move(0.0, -10.0); g_plot(0.0, 10.0);
    g_sel_line(1);
    /* */
    for (nstep = 0; nstep <= maxnstep; nstep++) {
        t = nstep * tau;
        /* u(・,t) の Fourier 係数を求める */
        at[0] = a[0];
        for (k = 1; k <= N; k++)
            at[k] = a[k] * exp(- k * k * t);
        /* 逆実 DFT により、関数値を求める。
         * 既に at[] には本来の Fourier 係数の 1/2 が入っているので、
         * cost() 呼び出しの直後に、すぐに関数値の列が出来上がる。 */
        cost(N+1, at, work);
        /* printf(" u(0,%f)=%f\n", t, at[0]); */
        /* グラフを描く */
        g_move(0.0, at[0]);
        for (j = 1; j <= N; j++) g_plot(j * h, at[j]);
    }
    g_sleep(G_STOP);
    return 0;
}

double f(double x)
{
    /* a0/2+a1 cos(x)+a2 cos(2x)+a3 cos(3x) */
    int i;
    static double a[4] = {1.0, 2.0, -3.0, 4.0};
    double result = a[0] / 2.0;
    for (i = 1; i < 4; i++)
        result += a[i] * cos(i * x);
    return result;
}
```

これを実行させると以下のようにな結果になる。

```
oyabun% ccx tstcost.c get_id.o -ldfftpack
oyabun% tstcost
N=128
Tmax=5.0
Ｂ=0.1
a[0]/2=       0.500000
a[1]/2=       1.000000
a[2]/2=      -1.500000
a[3]/2=       2.000000
a[4]/2=      -0.000000
oyabun%
```



3.2.5 実数値奇関数に対する DFT — 正弦変換

実数値奇関数に対する DFT — 正弦変換 —、のための手続きとして、

sinti sint で用いる定数表を初期化する。

sint $\{f_j\} \longmapsto \{B_k\}$ を計算する。その逆 $\{B_k\} \longmapsto \{f_j\}$ もこれで計算できる (後で正規化が必要になる)。

が用意されている。

sinti は **sint** を利用するために必要となる定数表を初期化するためのもので、

```
real wsave(2.5*MAXn+15)
```

```
double wsave[2.5*MAXn+15];
```

のように作業用配列 $wsave$ を宣言しておいてから、**sint** の呼び出しに先立って、

```
call sinti(n, wsave)
あるいは
call sinti(Nh-1, wsave)
```

```
sinti(n, wsave);
あるいは
sinti(Nh-1, wsave);
```

のように呼び出して、$wsave$ を初期化する必要がある。こうすると、この $wsave$ には、n に対応した定数表が格納されることになる。ここで n は **sint** の引数となる数列の長さで $N_h - 1$ に等しい。

さて、作業用配列以外には、関数値と Fourier 係数兼用の配列 x() が必要になる。これらは

```
real x(MAXn)
あるいは
real x(MAXNh-1)
```

```
double x[MAXn+1];
あるいは
double x[MAXNh];
```

のように宣言しておく。

Fourier 係数を求めるには、まず配列 x() に係数を求めようと思う実数値奇関数 f の、半周期区間 $[0, \pi]$ の N_h 等分点上での値を書き込む必要がある。これには、例えば

```
h = PI / Nh
do j=1,Nh-1
  x(j) = f(j*h)
end do
```

```
h = PI / Nh;
for (j = 1; j < Nh; j++)
  x[j] = f(j * h);
```

のようなコーディングをすればよい (もちろん PI は円周率 π である)。こうしておいてから、

```
call sint(n,x,wsave)
あるいは
call sint(Nh-1,x,wsave)
```

```
sint(n, x+1, wsave);
あるいは
sint(Nh-1, x+1, wsave);
```

のように **sint** を呼び出すと、配列 $x()$ に、順に

$$N_hB_1, N_hB_2, \cdots, N_hB_{N_h-1}$$

という N_h-1 個の数が格納される。つまり **sint** は 2.4 で説明した順方向実 DFT の N_h 倍を計算する。実 DFT そのものを計算したい場合は、 **sint** の呼び出しの後に[19]

```
do k=1,Nh-1
  x(k) = x(k)/Nh
end do
```

```
for (k = 1; k < Nh; k++)
  x[k] /= Nh;
```

のように割算をするわけである。

この逆に Fourier 係数から関数値の列を求めるにも **sint** を用いる。これは 2.4 で説明した逆方向離散正弦変換 の 2 倍を計算する。

きまじめに Fourier 解析に続いて、Fourier 合成を行なうには

```
call sint(Nh-1,x,wsave)
do k=1,Nh-1
  x(k) = x(k)/Nh
end do
何かの処理
call sint(Nh-1,x,wsave)
do j=1,Nh-1
  x(j) = x(j)/2.0
end do
```

```
sint(Nh-1, x+1, wsave);
for (k = 1; k < Nh; k++)
  x[k] /= Nh;
何かの処理
sint(Nh-1, x+1, wsave);
for (j = 1; j < Nh; j++)
  x[j] /= 2.0;
```

のようなプログラムとなるが、この「何かの処理」の中で、配列 $x()$ に対して行なわれる操作が、線形演算であるならば、二つの割算を一つの割算 ($2N_h$ で割る) にまとめること、例えば

```
call sint(Nh-1,x,wsave)
do k=1,Nh-1
  x(k) = x(k)/twoNh
end do
何かの処理
call sint(Nh-1,x,wsave)
```

```
sint(Nh-1, x+1, wsave);
for (k = 1; k < Nh; k++)
  x[k] /= twoNh;
何かの処理
sint(Nh-1, x+1, wsave);
```

のように出来る。もちろん $twoN_h$ は $2N_h$ という値をもつ変数である。

例題: 次の初期値・境界値問題を解くプログラムを作れ。

$$\text{(56)} \qquad \begin{cases} u_t(x,t) &=& u_{xx} & (x \in (0,\pi),\, t>0) \\ u(0,t) &=& u(\pi,t)=0 & (t>0) \\ u(x,0) &=& f(x) & (x \in [0,\pi]). \end{cases}$$

ただし $f(x) = 2\sin x - 3\sin 2x + 4\sin 3x.$

解答: 初期データ f を

$$\text{(57)} \qquad f(x) = \sum_{k=1}^{\infty} b_k \sin kx$$

のように Fourier 級数展開したとすると、

$$\text{(58)} \qquad u(x,t) = \sum_{k=1}^{\infty} e^{-k^2t} b_k \sin kx$$

で解が求まる。これを **sint** を使って解くプログラムを紹介しよう。

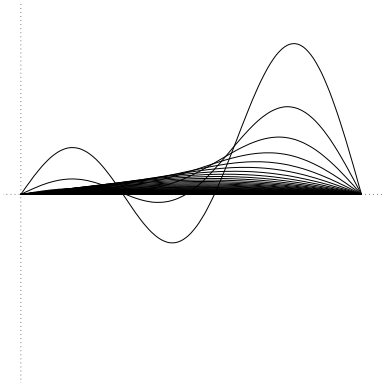
[19] 別に前にやっても構わない。

```
/*
 * tstsint.c -- test of sinti(), sint() in FFTPACK
 *   How to compile: ccmg tstsint.c get_id.c -ldfftpack
 */

#include <stdio.h>
#include <math.h>
#include <glsc.h>
#include "dfftpack.h"

#define MAXN    4096

double f(double);

int main(void)
{
    /* ユーザーが入力する値 */
    int N;
    double Tmax, tau;
    /* 定数 */
    double PI;
    /* プログラムの実行を通じて不変な値を持つ変数，関数宣言 */
    int twoN, maxnstep, getint();
    double h, f(), getdouble();
    /* 制御変数，作業用配列 */
    int j, k, nstep;
    double t;
    double b[MAXN+1], work[3*MAXN+15], bt[MAXN+1];

    /* 定数値の設定 */
    PI = 4.0 * atan(1.0);
    /* 分割数 N，追跡時間 Tmax，時間刻み幅 τ の決定 */
    printf("N=");    N    = getint();    if (N <= 0 || N > MAXN) return 0;
    printf("Tmax="); Tmax = getdouble();
    printf("τ=");   tau  = getdouble(); if (tau == 0.0) return 0;
    /* プログラムの実行を通じて不変な値を持つ変数 */
    twoN = 2 * N; maxnstep = Tmax / tau; h = PI / N;
    if (maxnstep < 0) {
        fprintf(stderr, "Tmax, τの値を見直して下さい\n");
        return 0;
    }
    /* 初期値 */
    for (j = 0; j <= N; j++) b[j] = f(j * h);
    /* N 項実 FFT の準備の後、順方向 DFT によって Fourier 係数を求める
     * sinti(),sint() の第一引数は、数列の長さであり、区間の分割数では
     * ないことに注意しよう。だから N ではなくて、N-1 を指定する。 */
    sinti(N-1, work);
    sint(N-1, b+1, work);
    /* sint() の出力を N でなく 2N で割っている。
     *  → 本当の Fourier 係数の 1/2 を得る。 */
    for (k = 0; k <= N; k++) b[k] /= twoN;
    /* Fourier 係数 (* 1/2) の最初の数項を表示 */
    for (k = 1; k < 5; k++)
        printf("b[%d]/2=%15f\n", k, b[k]);
    /* ウィンドウを開き、適当に座標を入れる */
    g_init("SINT", 140.0, 140.0);
```

```
    g_device(G_BOTH);
    g_def_scale(0,
                -0.2, 3.35, -10.0, 10.0,
                10.0, 10.0, 120.0, 120.0);
    g_sel_scale(0);
    /* 座標軸を描く */
    g_def_line(0, G_BLACK, 2, G_LINE_DOTS);
    g_def_line(1, G_BLACK, 1, G_LINE_SOLID);
    g_sel_line(0);
    g_move(-0.2, 0.0); g_plot(3.35, 0.0);
    g_move(0.0, -10.0); g_plot(0.0, 10.0);
    g_sel_line(1);
    /* */
    for (nstep = 0; nstep <= maxnstep; nstep++) {
        t = nstep * tau;
        /* u(・,t) の Fourier 係数を求める */
        bt[0] = bt[N] = 0.0;
        for (k = 1; k < N; k++)
            bt[k] = b[k] * exp(- k * k * t);
        /* 逆実 DFT により、関数値を求める
         * 既に bt[] には本来の Fourier 係数の 1/2 が入っているので、
         * sint() 呼び出しの直後に、すぐに関数値の列が出来上がる。 */
        sint(N-1, bt+1, work);
        /* グラフを描く */
        g_move(0.0, bt[0]);
        for (j = 1; j <= N; j++) g_plot(j * h, bt[j]);
    }
    g_sleep(G_STOP);
    return 0;
}

double f(double x)
{
    /* b1 sin(x)+b2 sin(2x)+b3 sin(3x) */
    int k;
    static double b[4] = {0.0, 2.0, -3.0, 4.0};
    double result = 0.0;
    for (k = 1; k < 4; k++)
        result += b[k] * sin(k * x);
    return result;
}
```

これを実行させると以下のようにな結果になる。

```
oyabun% ccx tstsint.c get_id.o -ldfftpack
oyabun% tstsint
N=128
Tmax=5.0
Ｂ=0.1
b[1]/2=       1.000000
b[2]/2=      -1.500000
b[3]/2=       2.000000
b[4]/2=      -0.000000
oyabun%
```



3.3 明治大学数学科ワークステーションでの FFTPACK の使い方

FFTPACK はファイルサーバー oyabun の /usr/local/lib/ にインストールされていて、FORTRAN や C プログラムのコンパイル時に “-l” オプションによる指定で、リンクして使うことが出来る。

- 単精度用のライブラリィを使うには “-lfftpack” を指定する。
- 倍精度用のライブラリィを使うには “-ldfftpack” を指定する。

例えば、

```
f77 test.f -lfftpack
cco test.c -ldfftpack -lm
ccx tstezfft.c -ldfftpack
```

のようにしてコンパイルするわけである。

なお、FFTPACK そのものは FORTRAN で書かれているので、FORTRAN プログラムから呼び出せるのは当然であるが、C 言語のプログラムからも呼び出せるように、インターフェイスを取る関数も含めてある。ソースは

```
ftp://oyabun/usr/local/lib/mathlib/FFT/fftpack/
```

にある。

4 がらくたノート (準備中)

色々書き加えなくては。

4.1 FFT のアルゴリズムの簡単な解説

原理は？どれくらい速くなるのか。

4.2 FFT と親戚のアルゴリズム

4.3 周期 2π でない周期関数に対する FFT

2 節では、周期 2π の周期関数に対する FFT を説明したが、2π でない周期を持つ周期関数に対する FFT はどうなるか？

4.4 多次元の FFT

多変数関数の DFT, FFT はどうやって計算するのか？

4.5 Fourier 変換の流儀

1. 伊藤清三「Lebesgue 積分入門」など

$$\widehat{f}(\xi) = \int_{\mathbf{R}^n} f(x)e^{-2\pi i\xi x}dx \tag{59}$$

$$\widetilde{g}(x) = \int_{\mathbf{R}^n} g(\xi)e^{2\pi i\xi x}d\xi \tag{60}$$

2.

$$\widehat{f}(\xi) = \frac{1}{(2\pi)^{n/2}}\int_{\mathbf{R}^n} f(x)e^{-i\xi x}dx \tag{61}$$

$$\widetilde{g}(x) = \frac{1}{(2\pi)^{n/2}}\int_{\mathbf{R}^n} g(\xi)e^{i\xi x}d\xi \tag{62}$$

3.

$$\widehat{f}(\xi) = \int_{\mathbf{R}^n} f(x)e^{-i\xi x}dx \tag{63}$$

$$\widetilde{g}(x) = \frac{1}{(2\pi)^{n}}\int_{\mathbf{R}^n} g(\xi)e^{i\xi x}d\xi \tag{64}$$

4. Numerical Recipes

$$\widehat{f}(\xi) = \int_{\mathbf{R}^n} f(x)e^{2\pi i\xi x}dx \tag{65}$$

$$\widetilde{g}(x) = \int_{\mathbf{R}^n} g(\xi)e^{-2\pi i\xi x}d\xi \tag{66}$$

5.

$$\widehat{f}(\xi) = \int_{\mathbf{R}^n} f(x)e^{i\xi x}dx \tag{67}$$

$$\widetilde{g}(x) = \frac{1}{(2\pi)^{n}}\int_{\mathbf{R}^n} g(\xi)e^{-i\xi x}d\xi \tag{68}$$

一つだけ結論「もう統一するのはあきらめよう。」

参考文献

[1] 大浦 拓哉, 「FFT (高速フーリエ・コサイン・サイン変換) の概略と設計法」, http://momonga.t.u-tokyo.ac.jp/~ooura/fftman/index.html